"Express Mail" mailing label number:

EV 335897198 US

METHOD AND SYSTEM FOR JOURNALING AND ACCESSING SENSOR AND CONFIGURATION DATA

Michael Primm

CROSS-REFERENCE TO RELATED APPLICATION(S)

[0001] The present application claims priority from U.S. provisional patent application no. 60/462,847, filed April 14, 2003, entitled "METHOD AND SYSTEM FOR JOURNALING AND ACCESSING SENSOR AND CONFIGURATION DATA," naming inventor Michael Primm, which application is incorporated by reference herein in its entirety.

FIELD OF THE DISCLOSURE

[0002] The present disclosure relates generally to methods and systems for journaling and accessing sensor and configuration data.

BACKGROUND

[0003] Collection and processing of sensor readings and other data in typical limited resource systems presents a number of significant implementation problems. Typical limited resource systems have limited memory and processing capabilities. Often, data is collected and produced during interrupt processing on the system, and is subject to very rapid change. In many cases, the timeliness of these changes is critical, and the duration of significant changes may be very short (milliseconds are not uncommon). As a result, a number of serious design problems emerge in such systems for collecting and processing data.

[0004] For example, typical systems often have difficulty quickly and efficiently reporting new sensor readings to collection systems. In addition, typical systems are often unable to prevent short-term (millisecond range) updates, which may be significant, from being "missed" by processing code, such as code for implementing value thresholds, or for reporting value updates across a network. Furthermore, certain typical systems are unable to allow "slow running" code,

such as notification routines for sending data through e-mail or FTP, to view data in a "stable" state, so that a set of related values can be reported or processed in a consistent state ("temporal integrity"). Also, these systems often have difficulty delivering multiple data values consistently to collection systems, so that processing code does not observe temporary data inconsistencies ("transactional integrity").

[0005] Many typical systems write and rewrite single data structures, typically with some sort of semaphore or other mutual exclusion mechanism, and rely on frequent polling by data monitoring applications to observe problems or report changes. Other systems work on a "process on update" model, where the act of updating the data also includes immediately calling the necessary processing code, so that the data update is handled, but limits how much processing can be done before another data update can be reported and handled.

[0006] Large scale data processing systems, such as distributed relational database systems (RDBMS), often face variations of problems, such as frequent update, multiple users with differing rates of processing, and significant integrity requirements. A common mechanism used to aid in addressing these problems in many DBMS systems is the use of a data journal.

[0007] Unfortunately, such classic database systems are not suitable for limited resource monitoring systems for a number of reasons. First, they typically require large database applications, along with large amounts of file system based storage. Secondly, their client interfaces are exclusively "user-mode" useable as opposed to being useable by interrupt mode or kernel mode application code. Lastly, typical database systems lack the real-time to near real-time responsiveness needed both for data update and data access by limited resource monitoring system applications. Accordingly, there is a need for an improved system and method of accessing sensor and configuration data.

SUMMARY

[0008] In a particular embodiment, the disclosure is directed to a system including a processor, a sensor interface responsive to the processor and memory responsive to the processor. The memory includes program instructions operable to direct the processor to implement a kernel-

mode device driver for manipulating a journal based data system associated with data received via the sensor interface.

[0009] In another embodiment, the disclosure is directed to a system including a memory including a plurality of variable definitions, a plurality of variable update records, and a plurality of context records. Each variable definition of the plurality of variable definitions has an associated variable and includes an oldest update field and a latest update field. Each variable update record has an associated variable and includes a variable value, a next update pointer, and a previous update pointer. The previous update pointer of a first variable update record associated with one variable points to the oldest update field of a variable definition associated with the one variable. The next update pointer of a second variable update record associated with the one variable points to the latest update field of the variable definition associated with the one variable. Each context record of the plurality of context records has an associated current timestamp field and a context update field pointing to a third variable update record of the plurality of variable update records.

[0010] In a further embodiment, the disclosure is directed to a method for accessing a value associated with a variable at a target time. The method includes searching a variable record table for a variable record. The variable record has a variable identification associated with the variable, a latest update pointer, a creation time not greater than the target time, and a latest update time. The method includes selectively searching a set of update records starting with a first update record indicated by the latest update pointer and following a previous update pointer included in the first update record to a subsequent update record.

[0011] In another embodiment, the disclosure is directed to a method for managing memory. The method includes determining an oldest timestamp of interest, searching a set of update records starting at a specified start point and proceeding chronologically to subsequent update records until identifying a first update record with a timestamp newer that the oldest timestamp of interest, setting the specified start point to the record chronologically following the first update record and revising a variable record associated with the first update record.

BRIEF DESCRIPTION OF THE DRAWINGS

[0012] **FIG. 1** is a block diagram depicting an exemplary monitoring system.

[0013] **FIG. 2** is a block diagram depicting an exemplary embodiment of a monitoring device.

[0014] **FIG. 3** is a block diagram depicting a table of variable records.

[0015] **FIG. 4** is a block diagram depicting links between a variable record and update records.

[0016] **FIG. 5** is a block diagram illustrating a global update record set.

[0017] **FIGs. 6 and 7** are flow diagrams illustrating exemplary methods for use by the system.

DETAILED DESCRIPTION

[0018] The present disclosure describes a method and system for utilization of data journaling techniques to produce a real-time sensor monitoring and data processing system suitable for use in embedded monitoring appliances or, alternatively, in systems, such as PCs and servers.

[0019] The term 'monitoring appliance', as used in this disclosure, generally refers to a low-cost computing device built to perform a well-defined set of functions centered around the acquisition and processing of analog and digital sensor readings. Unlike personal computers (PCs) and other large scale computing devices, monitoring appliances typically have limited storage and limited computing power. Storage generally includes RAM and Flash memory. In general, mechanisms for collecting and producing the sensor readings in an appliance involves significant interrupt mode and kernel mode processing unlike monitoring applications on PC devices. Applications on PC devices typically do the bulk of data collection and processing in processes in "user mode," where full operating system functionality is available. In particular examples, the monitoring appliance may include applications for alarming and data communication through networks using standards such as HTTP, FTP, SMTP, SMS, and SNMP.

[0020] The disclosure presents a memory and processing-based system for implementing a journaling style data model (which may not have a file system or high-level operating system functionality). A journal based data model includes a set of variable updates. Such a journaling

style data model may be used on a monitoring appliance. In one particular embodiment, the journaling style data model may be implemented in a kernel mode or as a device driver. Kernel-mode generally includes running with supervisor privilege within the kernel of the operating system. Device driver generally includes a specialized piece of code, either integrated with or loaded by the operating system kernel, that runs with kernel privilege and provides a service or access to a hardware device. In one exemplary embodiment, the system provides interfaces for interrupt, kernel, and user mode update and access to the data model, including transactional data update (with rollback) from interrupt, kernel, and user mode. The system may also include access to stable "snapshots" of the state of the data model as of a given time without copying the data model. The system may provide support for step-by-step traversal of the update sequence for the data model, allowing even slow applications to avoid missing short-term data events. Each step in the traversal provides a stable view of the whole data model at the point in time in the data model's history that corresponds to the time of the last update read. The system may provide support for efficiently reclaiming journal space as accessing applications move past the older updates. The system may also provide mechanisms for defining named variables of various data types, as well as supporting "meta-data" for these variables.

[0021] FIG. 1 depicts an exemplary monitoring system. A monitoring appliance 102 monitors a space 120 and, in some embodiments, equipment 110, such as computer equipment. The monitoring appliance 102 may communicate data associated with the space 120 and the equipment 110 to a remote system 112. For example, the monitoring appliance 102 may gather environmental data, such as temperature, air flow, humidity, leak detection, power quality, and motion detection.

[0022] The monitoring appliance 102 includes a journaling database system 104. The monitoring appliance 102 may also include or communicate with sensors 106 or cameras 108. The journaling system 104 may be a kernel-mode driver including a table of variable records, a plurality of variable updates, and a plurality of context records.

[0023] FIG. 2 depicts an exemplary monitoring appliance 202. The monitoring appliance 202 includes circuitry 204, sensor interfaces 206, network interfaces 216 and memory 218. Memory 218 includes a journaling database 222, various other data 214, and programs and software

instructions 220. The journaling database 222 includes a table of variable definitions 208, a set of variable update records 210, a set of context records 212. The circuitry 204 performs logic functions, data gathering, and communications functions and may include a processor and other support circuitry.

[0024] The sensor interfaces 206 may interact with sensors, such as analog or digital sensors. For example, the sensor interfaces 206 may gather temperature, humidity, airflow, video, audio, dry-contact, and other data from specific sensors. Network interfaces 216 may provide communications connectivity. For example, the monitoring appliance may communicate with wired and wireless networks, such as Ethernet, telephony, Blue Tooth®, 802.11, 802.16 and other networks.

[0025] Memory 218 generally takes the form of random access memory (RAM) and flash memory. However, permanent data storage may be provided. The memory includes tables and record sets 208, 210, and 212, which form the base data structures for the journaling data model 222. The table of variable definitions 208 contains records describing data objects within the data model. In a particular implementation, these records are arranged in a hash table, such as a table hashed using the name of each variable, to allow fast lookup. For faster access, the records describing each variable may be fixed in memory, allowing a pointer or some other index to be used as a handle for accessing a variable.

[0026] The global set of variable update records 210, also referred to as a global journal, includes an ordered journal of updates, providing a history of the value updates of sensor values. Each update record is also a member of a variable-specific update set, allowing the value history for a given variable to be easily accessed. In one particular embodiment, the global set or global journal 210 is implemented as a next update pointer in each update record.

[0027] The set of context records 212 describes "views" into the data model for each of the applications accessing the data model. Each context record may include a pointer into the global update journal and a corresponding update timestamp. The pointer indicates where in the history of the data model the context is referring. Context records may also include support for filtering the view of the data, for example, by restricting the view to selected variables that are of interest.

[0028] Memory 218 may also include other data 214 such as threshold data, application data, video data, audio data, alert and alarm contact data, and other data objects associated with performing monitoring functionality. Memory 218 may also include computer-implemented programs 220 configured to direct circuitry 204 to provide functionality, such as, for example, data monitoring, alert and alarm functionality, data transfer, and network communications. For example, the programs 220 may include a data monitoring application with data objects useful in comparing data to threshold values and useful in determining alarm conditions. The programs and instructions 220 may further include applications configured to contact, notify or alert other systems and personnel when alarm conditions occur. The programs and instructions 220 may further include applications, such as web servers and email applications, for transferring or communicating information and data. For example, the programs and instructions 220 may be configured to direct the circuitry 204 to communicate via the network interfaces 216 using communications standards, such as FTP, SNMP, SMTP, HTTP, and SMS.

[0029] Each variable record in the data model defines a uniquely named entity, typically using a textual name, with a specific data type. In a particular embodiment, each sensor in the system has an associated variable for which the current value is the sensor reading. Other data, such as configuration, error conditions, status, and security settings can be stored as variables. An exemplary structure for a variable record is represented in Table 1.

[0030] Table 1. Exemplary Data Structure

ulong Magic_Number (indicates type of structure, and to verify pointers)
Char *ID_string (unique ID of variable)
Class_ptr Class (pointer to "class" of variable – use for categorization)
Enum vartype Type (enumeration describing which data type is used by the variable)
ulong Creation_timestamp (used to resolve between different copies of a variable that might exist due to the variable being destroyed and recreated)
priv Privilege (access privileges)
varupdate Oldest_Update (update record containing the oldest available value of the variable)
varupdate *Latest_Update (pointer to most recent value update of the variable)
varmeta *Metadata_List (pointer to list of metadata records, providing extra variable data)
Enum varflags Type_Flags (used for basic attributes (read-only, persistence))
variable *Next_Variable (used for linking the record into lists within the hash table)
ulong History_Time (setting for controlling how much value history to preserve,

independent of journal length)
varhist *History_List (list of history records for holding values that have been removed from the journal but are still needed to support the History Time)

[0031] The variable records may be stored in or accessed via an open hash table, with the records hashed by the ID_string. As shown in FIG. 3, a hash table 302 may include records hashed by ID-string, such as records 304, 306, and 308. These records may point to update records with timestamps as indicated by records A, B, C, G, and FF.

[0032] Variable records referenced in the hash table may be uniquely identified by ID_string in combination with creation time. Creation time is useful in handling the situation in which a variable is deleted and recreated during the lifetime of the journal. When the hash table is searched for a variable, the lookup is relative to a desired time reference, for example, either the present time or the context used to view the data. The combination of the creation time and the time of the last update of a given variable is used to select from among the records that may exist for a given variable ID. If the variable is deleted, the time of the last update is the time of the deletion. Likewise, if no variable record existed at the desired time reference, which may be before or after the variable was created or destroyed, the lookup can recognize the absence of the variable record and act accordingly.

[0033] Each variable record may also include descriptive data, such as the data type, attribute flags, and class of the variable. In a particular embodiment, a significant variety of data types are supported, including signed and unsigned integers (32-bit and 64-bit), single and double precision floating point numbers, Booleans, strings, string lists, timestamps, byte arrays and structured data values, such as lists of other data value primitives with ID strings for each.

[0034] This exemplary embodiment also supports a variety of attribute flags including basic access control, such as a read-only flag, a constant flag, and an undeletable flag; name scope control, such as a flag indicating that the variable ID is globally unique across multiple systems or locally unique; and persistence control, such as a flag for indicating that the variable should be saved to permanent storage and restored when the system is restarted.

[0035] In an exemplary embodiment, each variable record also includes a variable update record. This variable update record stores the "oldest" value of the variable, which provides the base-line version of the variable. The updates contained in the journal are, generally, assumed to be relative to the base-line values stored in the variable records. For example, if no update is in the journal, or if the update is after the time in question, the "oldest" value embedded in the variable will be used to describe the value of the variable.

[0036] Variable values may be stored in variable update records. In one particular embodiment, each variable record consists of the fields depicted in Table 2.

[0037] Table 2. Exemplary Variable Record Fields

variable *var – the variable that the updated value refers to
ulong update_time – the timestamp that the value update occurred
varvalue value – the updated value
varupdate *next_var_update – pointer to the next newer update for the same variable
varupdate *prev_var_update – pointer to the next older update for the same variable
varupdate *next_global_update – pointer to the next newer update in the global journal

[0038] Variable update records may be linked into two different ordered sets. One set represents the value history of the specific variable, such as through using the next_var_update and prev_var_update pointers, and the other set represents the global value history or global journal for the variables in the system. In both cases, the order of the records may be sorted by update timestamp with records "before" a given record having a timestamp equal to or lower than that of a given record, and records "after" a given record having a timestamp equal to or higher than that of the given record. This ordering is naturally produced due to the insertion process, since newer updates will have higher timestamps than older ones.

[0039] In general, references to "timestamp" in this disclosure are intended to refer to numeric values representing an ordered sequence of time in history. Depending upon the implementation, timestamps may be literal timestamps with sufficient resolution so that matching values can be considered by the system to be "simultaneous" or an incrementing "version stamp" where each atomic update or set of updates receives a new value which is higher than all previous values, such as by using an incrementing counter. In general, the mechanism used for producing timestamps should satisfy the following exemplary rules:

- A single variable cannot have more than one update with a given timestamp value (i.e. any two updates for a given variable must have different timestamps).

- Any two updates to two different variables that are not considered to have occurred simultaneously must have different timestamps, with the older of the two updates having a timestamp lower than the newer one.

- Any two updates to two different variables that are considered to have occurred simultaneously must have the same timestamps.

[0040] For a given variable, the combination of the variable record and its update records form a doubly-linked circular list. For example, as shown in FIG. 4, a record for Variable X includes a Latest Update field that points to an Update Record at the most recent time, in this case t=100. The Update Record at t=100 points to the previous Update Record at t=56. The Update Record at t=56 points to the Update Record at t=100 and the Update Record at t=40. The Update Record at t=40 includes pointers directed at the Update Record at t=56 and the Oldest Update Record at t=25. The Oldest Update Record may be incorporated into the Variable Record X.

[0041] Similarly, the update records, including those embedded in the variable record, are linked and ordered into a single list, representing the global value journal for the whole data system. As shown in FIG. 5, the global journal start pointer indicates an update record with the oldest timestamp, in this case, the lowest time t=25. Each update record points to the next chronological update record. For example, an update record for a variable (X) at t=25 may point to an update record for a different variable (Y) at t=26. Similarly, an update record for a variable (Y) at t=26 may point to an update record for the same variable (Y) at t=28. Update records for different variables may have the same timestamp. The most recent update record is the journal end pointer, in this example, an update of variable Z at t=102.

[0042] In general, when the value of a variable is updated, a new update record is created and initialized with a new value, an appropriate timestamp, and pointer to the variable record. The new update is added to both the end of the global journal (using the "journal_end" pointer) and the newest end of the variables local set of values (using the 'latest update' pointer in the variable record). The simple structure of the journal allows this operation to be quickly

accomplished, even in interrupt mode with, for example, protection provided by interrupt disables, spin locks, or other system appropriate mechanisms. If interrupt mode update and/or access is not used, semaphores can also be used for providing concurrency protection.

[0043] Since the value history of a variable is likely to be of interest, deletion of a variable is treated as a special case of a value update. A special reserved value is stored in the value field of the update record, indicating that the current value is now "deleted". When the variable is being accessed for its value at a given point in time, the search of the hash table can quickly determine that a given variable record did not exist at a given point in time by comparing the desired timestamp to the creation time and to the value and timestamp of the latest update. If the creation time is newer than the timestamp, the variable did not exist yet. If the value is 'deleted' and the timestamp is before the desired time, the variable no longer existed at that time.

[0044] Stable reader access to the data model and journal may be accomplished using data records known as 'access contexts'. In a particular implementation, an access context consists of the fields depicted in Table 3.

[0045] Table 3.

Ulong current_ts – current timestamp of data being accessed through this context
enum varflags Type_Flags_Mask – bit-mask for selecting which 'Type_Flags' bits are required to match
enum varflags Type_Flags_Value – bit-mask for providing the values of the bits of 'Type_Flags' which were selected by 'Type_Flags_Mask' (i.e. Type_Flags AND Type_Flags_Mask == Type_Flags_Value for matching variables)
varclass *class – class of variables which will match (null if any variable class is OK)
varcontextmetafiler *meta_flt – list of metadata slot Ids and values which a variable must match to be selected (null if no metadata slot restrictions)
varupdate *next_update – pointer into journal to next update to be read/processed when reading updates sequentially: null if all updates have been read
varaccesscontext *next, *prev – pointers for listing all access contexts

The current_ts field is used to provide a target time when the value of a variable is requested and to prevent pruning of updates. The minimum value of the current_ts field for active access

contexts can be found, maintained, and used for setting the limit on what portion of the journal is pruned.

[0046] The 'next_update' pointer is used to point directly into the update journal, allowing updates to be read, one by one, from the journal. When it is combined with the current_ts, the system can determine when more than one update occurred simultaneously. The updates can be read, one-by-one, and some may be found to have the same update timestamp.

[0047] In a particular implementation, the other fields (Type_Flags_Mask, Type_Flags_Value, cls, meta_flt) are used to "filter" the variables shown through the access context to only those of interest. Each of these fields, when defined, provides matching requirements for different fields in the variable record to be matched in order for a variable to be "of interest" to the access context. The other fields may be used to allow quick traversal of the update journal. Updates associated with variables not matching the filters are skipped without being reported to the reader of the access context.

[0048] Enumeration of the variable population and associated values at the given time represented by the access context may be accomplished by traversing the hash table of the variable records, skipping records representing variables that do not exist at the target time and records that do not match the access context's filters, and reading the values using the current timestamp as the target time. For large variable populations with frequent filtering, additional hashing or sorting of variable records with common filtered attributes may be performed to increase search efficiency.

[0049] Accessing the value of a variable at a given time may be accomplished as shown in FIG. 6. As shown in step 602, the hash table is searched for the variable records to find the record with the matching ID for the variable and where the creation time is less-than-or-equal-to the target time, and where either the latest update value is not 'deleted' or the latest update time is greater-than-or-equal-to the target time. If no record is found, the variable does not exist. If found, the update record list is searched, starting at the update record pointed to by the 'latest update' pointer, until a record with a timestamp less-than-or-equal-to the target time is found, as shown in step 604. If found, this is the value of the variable at the target time. If the timestamp

of the 'oldest' update record, for example, in the variable record itself, is greater-than the target time, the value at the target time is no longer in the journal.

[0050] In a particular embodiment, the update list for a given variable is a simple, double-linked list, which may be traversed linearly to find a value. Linear traversal is generally desirable where the length of the update history for a given variable is small, timestamp comparisons are very fast, and most value queries are for either the current value (the first in the list) or one of the most recent values (those closest to the start of the search). In implementations where these assumptions may be inappropriate, use of a heap, binary tree, or other data structure providing a faster-than-linear lookup may be used.

[0051] Since the access contexts consist of a very small set of data, contexts can be easily created or copied. One common use of access contexts is for threshold processing. Threshold processing code may operate either by reading updates, one by one, and checking for threshold violations on those variables of interest that are changing values or by skipping forward by prescribed periods of time, and traversing all desired data. Checking one by one prevents the need to poll all variables frequently since the threshold code can assume that a variable representing a sensor still has the same value as the last time it was read unless a new value update is reported. If a problem is found, it is commonly the case that another program, such as a notification process for sending e-mail or generating reports, is invoked. By copying the access context and passing the copy to the notification program, the system provides that the notification process reports on the same system state, including the same sensor values and other configuration data, as present at the time that the problem was reported. Meanwhile, the threshold processing code can continue to process newer data updates, without waiting for the notification code in order to preserve the data state, and without causing the notification code to potentially report "newer" system state than intended. When the notification process is completed, it can close or release the access context, allowing the system to prune the journal of the older updates that are no longer of interest.

[0052] The journal supports being "pruned" based on retiring records in the journal older than the oldest timestamp of interest, as shown in FIG. 7. Starting at the "journal-start", each update record is checked until one is found that is newer than the oldest timestamp value still of interest,

as shown at step 700. In one exemplary embodiment, the oldest timestamp would be indicated by the context record with the oldest current_ts field. As shown in step 702, the update record is removed from the journal. The journal-start is reset to the update record found after the record, as shown in step 704. The age of the record is determined, as shown in step 706. If the update record is not the oldest record for the given variable (based on comparing its pointer to the address of the "oldest" update record within the variable record owning the update record), the contents of the update record are copied into the "oldest" record, as shown in step 708, the update record is removed from the variable's update set, as shown in step 710, and the update record is deleted, as shown in step 712. If the record was the 'latest' for the variable, the 'latest' pointer is directed to the 'oldest' record within the variable record. If the update record is the oldest for the given variable, and the value is 'deleted', as shown in step 714, the variable record is unlinked from the variable hash table, as shown in step 716, and the variable record is deleted.

[0053] Pruning of the journal can be initiated for a variety of reasons, including closing of data model readers, resulting in changes in the oldest timestamp of interest; readers moving forward through the journal also causing such changes; or resource constraints, such as the population of update records exceeding a system prescribed limit or age limit.

[0054] Another use of data stabilization is for potentially long-running data processing, such as saving the variable data to disk or accessing the data across a network. The time-consistency gained by the access context allows the "slowness" of these mechanisms to not cause problems with their successful execution either by forcing the sensors and other data "updaters" to block or stop updating temporarily, or by requiring the production of a memory expensive copy of the data into fast storage, such as RAM, so that the slow processing can have a stable data image on which to operate. The option for such processes to "listen" for just the values that have been updated, by, for example, traversing the update journal, also allows for efficient incremental data communication such as only sending the data updates across a network, instead of a full copy of mostly-unchanged data.

[0055] Variable records also support a mechanism for providing additional "supporting data" for the variables. Such "metadata" provide a means by which additional information on the

variables, such as units, value ranges, labels, access control, and other application-specific data can be "tied to" the variables.

[0056] In a particular implementation, each "metadata slot" is defined by a record containing the slot's ID string, its data type (supporting the same types as the variable data supports), and any access control information. Each "metadata slot value" contains a pointer to the record defining the slot, the value of the slot, and pointers for chaining with the other metadata slot values for a given variable.

[0057] Multi-update transactions may be implemented by accumulating a desired set of update records without adding them to the journal or the variables' update chains until the transaction is "committed". When committed, updates in the chain are given the same update timestamp, added to the global journal, and linked into each of the update chains for each of the update variables as if the updates were done one by one. Only the commit processing is performed within the "mutual exclusion" (interrupt disable or semaphore, as indicated earlier) section of the code. The transaction can be accumulated step-by-step without entering the critical section. Likewise, the transactions can be "rolled-back" by simply deleting the chain of updates before commit. Similarly, creation of new variable objects within a transaction is accomplished by accumulating the new variable records, and adding them (and setting the creation timestamp) when the commit is complete.

[0058] In a particular implementation, each variable can optionally be assigned to be of a certain class. Variable classes are defined by data structures that include a unique ID string, a pointer to a parent class (if any), access privileges, and a list of class-specific metadata values. Exemplary Classes are used in the following ways:

1. Variable classes can provide metadata slot values, which can be treated as the default value for any slots not specifically defined by a given variable. This allows significant memory savings when metadata is frequently used, and often common for different variables of the same class. Since classes support having "parent classes", this inheritance of slot values can be easily extended: new subclasses can provide overriding slot defaults while simply inheriting other defaults.

2. Classes and superclasses can be efficiently used for filtering among different variables when enumerating or when scanning updates using an access context, since the "instance-of" operation can be implanted by a simple "pointer compare" combined with a "tree walk" from the variables class, to its parent, etc.

3. Defining access control

4. Providing for registering listener functions, such as constructors, destructors, update verifiers, and update notifiers.

[0059] The exemplary implementation includes multiple access methods for the data model implementation. Exemplary methods of access include implementation of these functions:

1. Creating, updating, and deleting variables and their metadata
2. Defining data classes and metadata slots
3. Creating, copying, and deleting access contexts, and setting filters on those access contexts.
4. Enumerating and querying variables using an access context (or accessing the "current values" without one)
5. Watching and processing updates, step-by-step.

[0060] An exemplary implementation is based around a kernel-mode device driver and provides a set of C-callable function calls providing the full set of data model capabilities. A kernel-mode implementation generally means running with supervisor privilege within the kernel of the operating system. A device driver generally means a specialized piece of code, either integrated with or loaded by the operating system kernel, that run with kernel privilege and provides a service or access to a hardware device. These interfaces are designed to be called either from interrupt-context or task-mode, allowing their use from other kernel-mode device drivers and subsystems.

[0061] A particular exemplary implementation provides a character device driver interface, including a full set of ioctl()-based functions, for implementing the full set of data model

capabilities. In addition, the character driver includes support for asynchronous (non-blocking) I/O, allowing select() to be used for timed waiting for updates from an access context.

[0062] In addition, a more friendly C-style interface may be provided through a shared library (which may be implemented using the character driver interfaces). An object-based C++ style interface may be used for implementation, as well. To allow easy use from scripting environments, several command-line interface tools may be implemented using the C-style library, providing access and modification mechanisms. The exemplary implementation may also include a set of CGI (Common Gateway Interface) modules implementing both query and update XML grammars for the data model. The Appendix depicts an exemplary XML grammar. These grammars may allow a full set of methods for querying the data model and returning subsets of the data model, as well as creating, modifying, and deleting data within the model. These XML-based web interfaces may provide a programmatically friendly way for a Java-based application provided as a GUI implementation, a centralized mass administration tool, and other third party applications to interact with the device implementing the data model. A query CGI may include support for both inputting an XML-based query (as form-data input of a POST) and returning the data in the response, or running pre-canned queries stored on the device and tied to defined URLs. An exemplary implementation may also include support for substitution of values derived from the data model into standard HTML documents using an XML-based macro grammar.

[0063] An exemplary implementation may include a custom SNMP subagent, which provides a mechanism for defining tables to the SNMP agent representing the values and metadata slots of variables of a given class. These mappings may provide a means by which variables in the data model, such as sensors and configuration data, can be easily published for access through standard SNMP access mechanisms.

[0064] Another exemplary implementation may include mechanisms for generating XML-based reports (using the same grammars as used for the Web Access), and delivering these reports using FTP to an FTP server, HTTP POST-ing (form data delivery) to a remote web server, or SMTP-based e-mail delivery.

[0065] A further exemplary implementation may include an application for utilizing the C-style library interface for monitoring data updates, and storing copies of the variables marked as persistent (using the type flags) to an XML-based file encoding. At system restart, this file may be used to drive the data model updates to restore those variables to the state corresponding to the last stored state. This same mechanism can be used for saving and restoring configuration data externally from the device.

[0066] The above disclosed subject matter is to be considered illustrative, and not restrictive, and the appended claims are intended to cover all such modifications, enhancements, and other embodiments which fall within the true scope of the present invention. For example, the above data model and journaling application may be implemented on PCs and server systems. Thus, to the maximum extent allowed by law, the scope of the present invention is to be determined by the broadest permissible interpretation of the following claims and their equivalents, and shall not be restricted or limited by the foregoing detailed description, including the APPENDIX.

APPENDIX

XML Grammar (DTD)

```
<!-- Data set schema: used for representing query
  results and presistent data sets.  The encoding is
  used to represent the definitions, values, and
  metadata of a set of variables defined in the
  nbVariables data manager -->

<!ELEMENT variable-set ((classdef|variable|query-error)*)>
<!ATTLIST variable-set
        privset CDATA #IMPLIED
        timestamp CDATA #IMPLIED
        time CDATA #IMPLIED>
<!-- variable class element -->
<!ELEMENT classdef (metadata*)>
<!ATTLIST classdef
        class CDATA #REQUIRED
        parent CDATA ""
        readpriv CDATA "0"
        writepriv CDATA "0">
<!-- variable instance element -->
<!ELEMENT variable ((null | i32-val | u32-val | i64-val |
                u64-val | float-val | double-val | bool-val |
                utc-val | utc-msec-val | string-val | password-val | octet-val |
                string-list-val | nls-string-val | nls-string-list-val |
                varid-val | varid-list-val | struct-val), metadata*)>
<!-- variable attributes:
                varid is LUID of variable (unless globalid="yes")
                class is class ID string -->
<!ATTLIST variable
        varid CDATA #REQUIRED
        guid CDATA #IMPLIED
        class CDATA ""
        classpath CDATA ""
        readonly (yes | no) "no"
        constant (yes | no) "no"
        persistent (yes | no) "no"
        val-transient (yes | no) "no"
        remote (yes | no) "no"
        globalid (yes | no) "no"
        nodelete (yes | no) "no">

<!-- null type value -->
<!ELEMENT null-val EMPTY>
```

```
<!-- 32-bit integer value : value is decimal integer string -->
<!ELEMENT i32-val (#PCDATA)>
<!ATTLIST i32-val
        isnull (yes | no) "no">

<!-- 32-bit unsigned integer value : value is decimal integer
        string -->
<!ELEMENT u32-val (#PCDATA)>
<!ATTLIST u32-val
        isnull (yes | no) "no">

<!-- 64-bit integer value : value is decimal integer string -->
<!ELEMENT i64-val (#PCDATA)>
<!ATTLIST i64-val
        isnull (yes | no) "no">

<!-- 64-bit unsigned integer value : value is decimal integer
        string -->
<!ELEMENT u64-val (#PCDATA)>
<!ATTLIST u64-val
        isnull (yes | no) "no">

<!-- 32-bit float value : value is decimal floating-point
        string -->
<!ELEMENT float-val (#PCDATA)>
<!ATTLIST float-val
        isnull (yes | no) "no">

<!-- 64-bit float value : value is decimal integer string -->
<!ELEMENT double-val (#PCDATA)>
<!ATTLIST double-val
        isnull (yes | no) "no">

<!-- boolean value : value is in val attribute -->
<!ELEMENT bool-val EMPTY>
<!ATTLIST bool-val
        val (true | false) "false"
        isnull (yes | no) "no">

<!-- 32-bit utc time value : value is decimal integer
        string -->
<!ELEMENT utc-val (#PCDATA)>
<!ATTLIST utc-val
        isnull (yes | no) "no">
```

```
<!-- 64-bit utc milliseconds value : value is decimal integer
        string -->
<!ELEMENT utc-msec-val (#PCDATA)>
<!ATTLIST utc-msec-val
        isnull (yes | no) "no">

<!-- string value - value is string -->
<!ELEMENT string-val (#PCDATA)>
<!ATTLIST string-val
        isnull (yes | no) "no">

<!-- password value - value is string -->
<!ELEMENT password-val (#PCDATA)>
<!ATTLIST password-val
        isnull (yes | no) "no">

<!-- octet string value - value is sequence of hex digits (2 per byte)
        string (i.e. F002CD) -->
<!ELEMENT octet-val (#PCDATA)>
<!ATTLIST octet-val
        isnull (yes | no) "no">

<!-- string list value - value is list of <string-val> -->
<!ELEMENT string-list-val (string-val*)>
<!ATTLIST string-list-val
        isnull (yes | no) "no">

<!-- NLS string value - value is string -->
<!ELEMENT nls-string-val (#PCDATA)>
<!ATTLIST nls-string-val
        isnull (yes | no) "no"
        raw CDATA #REQUIRED>

<!-- NLS string list value - value is list of <nls-string-val> -->
<!ELEMENT nls-string-list-val (nls-string-val*)>
<!ATTLIST nls-string-list-val
        isnull (yes | no) "no">

<!-- variable ID value - value is string variable ID  -->
<!ELEMENT varid-val (#PCDATA)>
<!ATTLIST varid-val
        isnull (yes | no) "no">

<!-- variable ID list value - value is list of <varid-val> -->
<!ELEMENT varid-list-val (varid-val*)>
<!ATTLIST varid-list-val
```

```
        isnull (yes | no) "no">

<!-- structure value - value is list of <struct-element> -->
<!ELEMENT struct-val (struct-element*)>
<!ATTLIST struct-val
        isnull (yes | no) "no">

<!-- structure element -->
<!ELEMENT struct-element (null | i32-val | u32-val | i64-val |
        u64-val | float-val | double-val | bool-val |
        utc-val | utc-msec-val | string-val | password-val | octet-val |
        string-list-val | nls-string-val | nls-string-list-val |
        varid-val | varid-list-val | struct-val)>
<!ATTLIST struct-element
        fieldid CDATA #REQUIRED>

<!-- metadata record -->
<!ELEMENT metadata (null | i32-val | u32-val | i64-val |
        u64-val | float-val | double-val | bool-val |
        utc-val | utc-msec-val | string-val | password-val | octet-val |
        string-list-val | nls-string-val | nls-string-list-val |
        varid-val | varid-list-val | struct-val)>
<!ATTLIST metadata
        slotid CDATA #REQUIRED
        isclassdef (yes | no) "no">

<!-- query root element -->
<!ELEMENT variable-query ((result-filter | id-query | type-class-query)*)>

<!-- Result filter - controls what data from variables are returned -
        filter values apply until next result-filter tag is encountered.
        incvalue="yes" means include the value of each matching object
        incclass="yes" means include the "class=" attribute for each object
        incclasspath="yes" means include the "classpath=" attribute for each
        inctypeflags="yes" means include the type flag attributes (persistent, readonly)
        incguid="yes" means include the guid= parameter for each variable
        resolvenls="yes" means include the locale-resolved version of any nls-string
        follow-varid-val="yes" means also get any objects referenced by the varid or
                varid-list typed value of a matching object -->
<!ELEMENT result-filter (incmeta*)>
<!ATTLIST result-filter
        incvalue (yes | no) "yes"
        incclass (yes | no) "no"
        incclasspath (yes | no) "yes"
        inctypeflags (yes | no) "no"
        incguid (yes | no) "no"
```

```
        resolvenls (yes | no) "yes"
        follow-varid-val (yes | no) "no">

<!-- Result filter tag for requesting specific metadata slot - if slotid
        is '*' or empty, request is for all metadata slots. The type of the
        slot requested can also be specified. If the slot is a varid or
        varid-list and "follow-slot" is "yes", the variable that is
        referenced by the slot should also be returned.  -->
<!ELEMENT incmeta EMPTY>
<!ATTLIST incmeta
        slotid CDATA "*"
        type (null | i32 | u32 | i64 | u64 | float | double | bool |
                utc | utc-msec | string | password | octet | string-list | nls-string |
                nls-string-list | varid | varid-list | struct | any) "any"
        incclassdef (yes | no) "yes"
        follow-slot (yes | no) "no">

<!-- Request for specific variable (with given varid: varid can be
        a space or comma separated list of IDs) -->
<!ELEMENT id-query EMPTY>
<!ATTLIST id-query
        varid CDATA #REQUIRED
        realtime (yes | no) "no">

<!-- Request all variables with given class and given type flag values
        Nested <metadata> tags can be used to provide additional "must match"
        constraints -->
<!ELEMENT type-class-query (metadata*)>
<!ATTLIST type-class-query
        class CDATA ""
        readonly (yes | no | any) "any"
        constant (yes | no | any) "any"
        persistent (yes | no | any) "any"
        val-transient (yes | no | any) "any"
        remote (yes | no | any) "any"
        globalid (yes | no | any) "any"
        nodelete (yes | no | any) "any"
        realtime (yes | no) "no">

<!-- <query-error> is used to report processing errors in the <variable-set>
        response to a <variable-query> request.  Zero or more can be returned
        inline with other successful results. error-code attribute is used
        for error condition identifier string (nls-neutral).  raw is used to
        report a "raw" version of the error message (if available), in the same
        format as used for nls-string-val tokens.  -->
<!ELEMENT query-error (#PCDATA)>
```

```
<!ATTLIST query-error
        error-code CDATA #REQUIRED
        raw CDATA "">

<!-- <variable-write> is the root element for variable add/modify/delete
        requests. All the updates withing the <variable-update> are applied
        as a single transaction. <classdef> is used to add new class definitions.
        'results' attribute is used to control if the response to the <variable-write>
        (a <variable-write-results> document) should include all results, only
        errors, or no results at all. rollback-on-error is used to control whether
        or not the whole transaction should be cancelled (rolled-back) if any error is
        reported -->
<!ELEMENT variable-write ((variable-add | variable-update | variable-delete |
                meta-add | meta-update | meta-delete | classdef)*)>
<!ATTLIST variable-write
        result (all | only-errors | none) "all"
        rollback-on-error (yes | no) "no">

<!-- <variable-add> is used to create a new variable instance: the
        updateifexists attribute can be used to do an update instead when the
        variable already exists. -->
<!ELEMENT variable-add (variable*)>
<!ATTLIST variable-add
        updateifexists (yes | no) "no">

<!-- <variable-update> is used to update existing variables: the class and
        typeflags in the <variable> elements are ignored. Also, the type of
        the data must match the existing variable. Any metadata slots defined
        are added if they don't exist and updated if they do.
        if mergestruct=yes, update to structures are merge updates
        (field-level add/update), else structure is replaced -->
<!ELEMENT variable-update (variable*)>
<!ATTLIST variable-update
        mergestruct (yes | no) "no">

<!-- <variable-delete> is used to delete a variable and its associated
        metadata -->
<!ELEMENT variable-delete EMPTY>
<!ATTLIST variable-delete
        varid CDATA #REQUIRED>

<!-- <meta-add> is used to add a new metadata slot to an existing variable.
        The updateifexists attribute controls whether the slot should be
        updated if it already exists -->
<!ELEMENT meta-add (metadata*)>
<!ATTLIST meta-add
```

```
        varid CDATA #REQUIRED
        updateifexists (yes | no) "no">

<!-- <meta-update> is used to update an existing slot of an existing
        variable. -->
<!ELEMENT meta-update (metadata*)>
<!ATTLIST meta-update
        varid CDATA #REQUIRED>

<!-- <meta-delete> is used to delete an existing slot of an existing
        variable. -->
<!ELEMENT meta-delete EMPTY>
<!ATTLIST meta-delete
        varid CDATA #REQUIRED
        slotid CDATA #REQUIRED
        type (null | i32 | u32 | i64 | u64 | float | double | bool |
                utc | utc-msec | string | password | octet | string-list | nls-string |
                nls-string-list | varid | varid-list | struct) #REQUIRED>

<!-- <variable-write-results> is the root element for reporting the
        results of a <variable-write> command sequence -->
<!ELEMENT variable-write-results (variable-write-result*)>

<!-- <variable-write-result> is used to report the result of a single
        add/update/delete operation.  Note that some operations (such
        as any operation that modifies a variable and its metadata values)
        are actually a sequence of operations (a variable add/update plus
        one meta-add (update-if-exists=yes) for each metadata slot). -->
<!ELEMENT variable-write-result EMPTY>
<!ATTLIST variable-write-result
        op (variable-add | variable-update | variable-delete |
                meta-add | meta-update | meta-delete | classdef | parse |
                transaction) #REQUIRED
        varid CDATA ""
        slotid CDATA ""
        slottype (null | i32 | u32 | i64 | u64 | float | double | bool |
                utc | utc-msec | string | password | octet | string-list | nls-string |
                nls-string-list | varid | varid-list | struct) "null"
        class CDATA ""
        result (success | does-not-exist | already-exists | not-permitted |
                type-mismatch | read-only | parse-error | general-error |
                rollback) "success">
```